FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Performance of a Parallel Matrix Multiplication Routine on an Intel iPSC/860

Inge Gutheil, Werner Krotz-Vogel*

KFA-ZAM-IB-9308

Mai 1993
(Stand 07.05.94)

(*) PALLAS, Gesellschaft für parallele Anwendungen und Systeme mbH
Hermülheimer Str. 10, D-5040 Brühl

Parallel Computing 20 (1994), pp. 953-974, 1994

1.0 Performance of a Parallel Matrix Multiplication Routine on Intel iPSC/860

Inge Gutheil
Research Centre Jülich (KFA)
Central Institute for Applied Mathematics
D-52425 Jülich

Werner Krotz-Vogel
PALLAS
Gesellschaft für parallele
Anwendungen und Systeme mbH
Hermülheimer Str. 10
D-50321 Brühl

Abstract:

The performance of a parallel matrix-matrix-multiplication routine with the same functionality as DGEMM of BLAS3 was tested for different numbers of nodes on a 32-node iPSC/860. The routine was then tuned for maximum performance on this particular computer system. Small changes in the original code lead to substantially higher performance and in all tested configurations there is a critical matrix size $n \approx 50 \cdot np$, the number of processors, above which Intel's non-blocking *isend* is more efficient than the blocking *csend*. This shows that special tuning for a single machine pays off for large matrices.

1.1 Introduction

On today's distributed memory multiprocessors usually specialized computer scientists develop specific programs for a few special problems. The average natural scientist avoids porting his code to these machines. To reduce “startup time”[1] for the end user, we need portable parallel libraries.

The parallel linear algebra library SLAP[2] (SUPRENUM Linear Algebra Package) [6, 11, 12] was developed in the course of the German supercomputer project SUPRENUM [13]. The parts of this library performing parallel BLAS2 and BLAS3 [4, 5] operations and the solution of linear equation systems were later on changed to use the PARMACS library [2, 3, 9] instead of

---

1 learning and code rewriting

2 The SLAP referred to in this paper has *nothing* to do with the *Sparse Linear Algebra Package* SLAP.

SUPRENUM FORTRAN message passing. This work was carried out in collaboration between SUPRENUM GmbH and the German National Research Center for Computer Science, GMD.

Since March 1991, PALLAS GmbH, a German software company for high performance computing, has acquired SLAP. Those parts of SLAP which have been modified to use the PARMACS library are now distributed by PALLAS GmbH under the name of Scientific Linear Algebra Package (SLAP) [10]. They can be used on a variety of computers where the PARMACS library is supported, amongst them the Intel iPSC/860.

SLAP was originally designed as a library of linear algebra routines to be used on distributed memory multiprocessors (DMMP), using the single program multiple data (SPMD) programming paradigm. It uses a process structure, in most cases a ring of processes, irrespective of the underlying processor structure. It is assumed that each process can communicate with any other process. The processes are mapped to the processors by a host program. Usually only one process is assigned to each processor. The routines have to be called from the node program with names according to the LAPACK [1] naming conventions with an extension for the parallel version. Additional calling parameters describing the parallel environment follow those of the sequential LAPACK routine.

To allow consistent use of any matrix in different SLAP subprograms without any redistribution, all matrices are distributed to processes as follows: The processes are assumed to be arranged in a chain. A matrix is distributed to the local memories as blocks of columns, with adjacent blocks being assigned to neighboring processes. The user chooses the range of columns for each process. An integer array containing the block description is passed to the routines along with the numerical arrays (see [10]).

For performance reasons the SLAP subprograms call the sequential BLAS wherever possible [12]. The communication was designed with the “compute and send ahead” strategy in mind [6], i.e. data is sent as soon as possible and received as late as necessary.

To determine the performance that can be achieved by such a library, we measured the performance of the parallel BLAS3 matrix multiplication routine DGEMMP from SLAP on $np = 4, 8, 16, 32$ processors of an iPSC/860. We compared the results to the performance of the sequential BLAS3 routine DGEMM of the “Classpack Basic Math Library” written by Kuck & Ass. and delivered by Intel [8] for single-node computation on the iPSC/860.

1.2 Measuring Performance on iPSC/860

The iPSC/860 has no hardware performance monitor, in contrast to systems such as the CRAY. There is no global clock information, either, because each processor has its local system clock and the processors run independently. The only tool to measure performance is the routine *dclock* which, on each processor, measures wall clock time since system boot [7].

To measure the performance of a parallel program, information is needed on how long it took for the routine to finish on all processors. We implemented the program for performance measurement of DGEMMP in the following way:

Each processor creates its local part of the input matrices *A*, *B*, and *C* according to the dimensions read in and distributed to the processors by the host program. Thereafter, the processors are synchronized by a call to the PARMACS command BARRIER or iPSC *gsync*. After synchronization each processor calls *dclock* for the first time, and then all processors call DGEMMP and start the matrix multiplication at the same time. Immediately after DGEMMP is completed, *dclock* is called again on each processor. All processors compute the elapsed time and send it to the host which determines the maximum of the elapsed times on all processors.

To calculate the performance (in MFLOPS) we divided the total number of floating point operations necessary for a sequential matrix multiplication of the same size by the largest elapsed time. That means the whole time spent in DGEMMP on the slowest processor was used to compute the MFLOPS rate and only those floating point operations which are really necessary for matrix multiplication are counted.

Measuring efficiency for the parallel routine reveals another problem: Usually the efficiency *e* of a parallel program is defined by

$$e = \frac{T_1}{np \cdot T_{np}} = \frac{MFLOPS_{np}}{np \cdot MFLOPS_1}$$

where *np* is the number of processors used, T_1 is the time for the sequential program on one processor, and T_{np} is the time for the parallel program on *np* processors, $MFLOPS_1$ is the MFLOPS rate achieved on one processor, $MFLOPS_{np}$ is the MFLOPS rate achieved on *np* processors. As explained below (see "Results") T_1 can be measured only for square matrices with sizes up to $n = 800$ whereas for $np > 4$ T_{np} can be measured for larger matrices. As $MFLOPS_1$ increases only slightly for $n < 300$ and remains almost constant for $n > 400$ we decided to denote by $\max MFLOPS_1$ the maximum MFLOPS rate achieved on one processor and defined the efficiency for matrices with $n > 800$ as if $MFLOPS_1 = \max MFLOPS_1$ for $n > 800$, i.e.

$$e = \frac{MFLOPS_{np}}{np \cdot \max MFLOPS_1} \quad \text{for} \quad n > 800 .$$

1.3 Realization of DGEMMP

DGEMMP computes $C = \alpha A^{[T]} B^{[T]} + \beta C$ for distributed matrices $A^{[T]} \in \mathbb{R}^{m,k}$, $B^{[T]} \in \mathbb{R}^{k,n}$ and $C \in \mathbb{R}^{m,n}$ and scalars α and β which are known to all processes[3].

DGEMMP allows four cases for the input matrices which are all treated differently:

Case 1: Neither *A* nor *B* have to be transposed.
Case 2: Only *A* has to be transposed.
Case 3: Only *B* has to be transposed.
Case 4: Both *A* and *B* have to be transposed.

---

3 [T] here means that transposition of the matrix is optional.

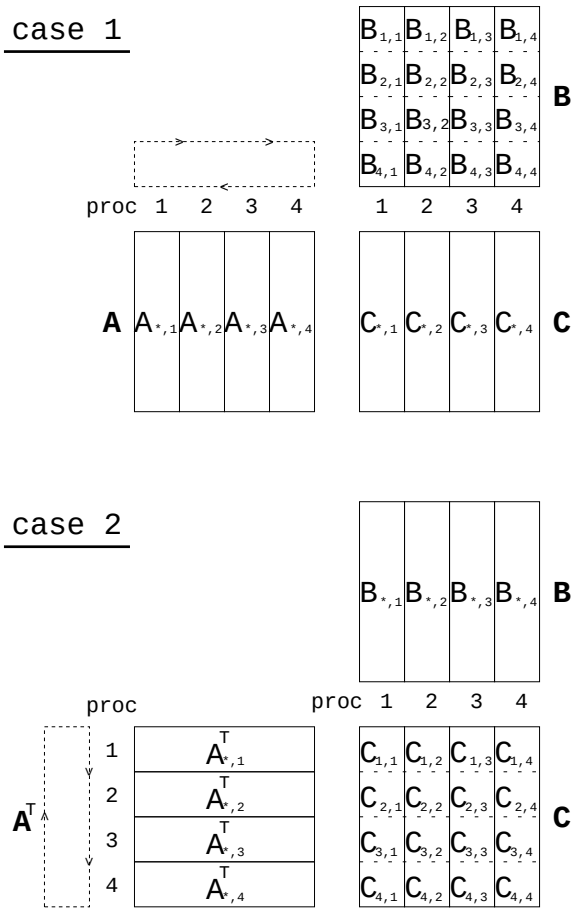

Fig. 1. Distribution of matrices to processes in cases 1 and 2: The dotted lines with arrows indicate that the blocks of the matrix are sent around the ring of processes.

Let the processes be logically arranged in a ring and numbered from 1 to *np*. Let $A_{*,i}$, $B_{*,i}$, and $C_{*,i}$ be the column blocks of *A*, *B*, and *C* respectively which process *i* has in its local memory on entry to DGEMMP. We refer to these as the local parts of the matrices. The SLAP concept requires that *A*, *B*, and *C* are column block distributed and not A^T or B^T. So if *A* has to be transposed ($A \in \mathbb{R}^{k,m}$ now), $A_{*,i}$ still is the column block of *A* which process *i* has in its local memory. Let $A^T_{*,i}$ denote the transpose of the column block $A_{*,i}$ of *A*, then $A^T_{*,i}$ is the row block of A^T process *i* owns.

In those cases, where *B* needs no transposition let $B_{j,i}$ and $C_{j,i}$ be those rows of $B_{*,i}$ and $C_{*,i}$ respectively with the same indices as the columns of *A* which process *j* owns on entry to DGEMMP. If *B* has to be transposed let $B_{j,i}$ be those rows of $B_{*,i}$ with the same indices as the columns of *C* which process *j* owns on entry to DGEMMP (see Fig. 1 and Fig. 2).

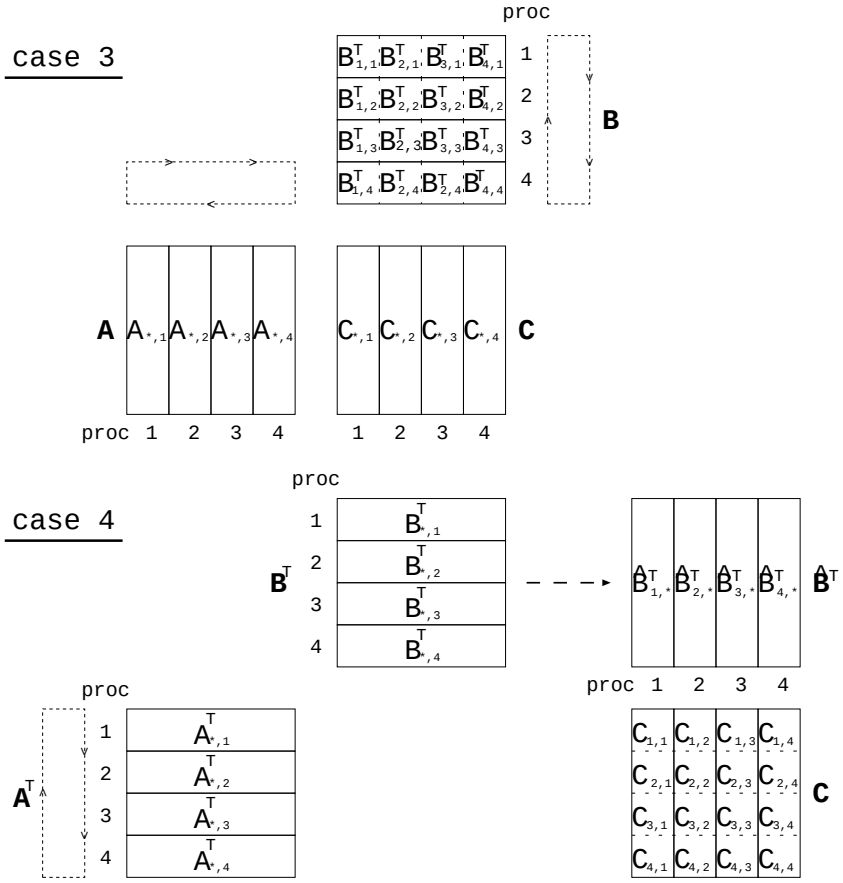

Fig. 2. Distribution of matrices to processes in cases 3 and 4: The dotted lines with arrows indicate that the blocks of the matrix are sent around the ring of processes.

Example: Let $np = 4$ and all matrices square with size $m = n = k = 100$. If all matrices are uniformly and equally distributed across the processes then $B_{*,1}$ denotes columns 1 to 25 of the matrix B. $B_{3,1}$ denotes rows 51 to 75 of columns 1 to 25 of matrix B in all cases. This is a square matrix of size 25×25.

In cases 1 and 2 process i computes its local part of C, $C_{*,i}$, by multiplying A by the local part of B, $B_{*,i}$. The matrices B and C therefore have to be distributed in the same way. As each process only has its local part $A_{*,i}$ of A in its memory communication is necessary. The computation and communication is done in np steps.

The algorithms in pseudocode:
(All indices or process numbers i have to be taken as $(i - 1) \mod np + 1$ to fit into the interval $[1, np]$.)

me = my_process_number

Case 1

Do k=1,np

If $k > 1$ *Receive* $A_{*,\,me-k+1}$ *from process* $me-1$

If $k < np$ *Send* $A_{*,\,me-k+1}$ *to process* $me+1$

Call DGEMM *to compute*
$$\begin{cases} C_{*,me} = \alpha A_{*,me} \quad B_{me,\,me} + \beta C_{*,\,me}, & \text{if } k = 1 \\ C_{*,\,me} = \alpha A_{*,\,me-k+1} B_{me-k+1,\,me} + C_{*,\,me}, & \text{else} \end{cases}$$

Enddo

Case 2

Do k=1,np

If $k > 1$ *Receive* $A_{*,\,me-k+1}$ *from process* $me-1$

If $k < np$ *Send* $A_{*,\,me-k+1}$ *to process* $me+1$

Call DGEMM *to compute* $C_{me-k+1,\,me} = \alpha A^T_{*,\,me-k+1} B_{*,\,me} + \beta C_{me-k+1,\,me}$

Enddo

In the third case where A needs no transposition but B has to be transposed in the original SLAP version of DGEMMP not only the local parts $A_{*,i}$ of A are sent around the ring but also parts of $B_{*,i}$. In the first step, process i sends all rows of $B_{*,i}$ it does not need itself. In the k-th step it sends those parts of the received $B_{*,i-k+1}$, which it does not need itself, to its neighbor. So a decreasing number of submatrices of $B_{*,i}$ is sent around the ring along with $A_{*,i}$. In this case, it is necessary that B is distributed in the same way as A.

The algorithm in pseudocode:

me = my_process_number

Case 3

Do k=1,np

If $k > 1$ *Receive* $A_{*,\,me-k+1}$ *and* $B_{j,\,me-k+1}$ *for* $j \notin \{me-l \mid 1 \le l < k\}$ *from process* $me-1$

If $k < np$ *Send* $A_{*,\,me-k+1}$ *and* $B_{j,\,me-k+1}$ *for* $j \notin \{me-l \mid 0 \le l < k\}$ *to process* $me+1$

Call DGEMM *to compute*
$$\begin{cases} C_{*,\,me} = \alpha A_{*,\,me} \quad B^T_{me,\,me} + \beta C_{*,\,me}, & \text{if } k = 1 \\ C_{*,\,me} = \alpha A_{*,\,me-k+1} B^T_{me,\,me-k+1} + C_{*,\,me}, & \text{else} \end{cases}$$

Enddo

Case 4, where A and B have to be transposed, is treated in a different way. Here the matrix B is redistributed to a working matrix $\hat{B}$ block row wise by a call to a SLAP auxiliary routine. $\hat{B}$ is the same matrix as B. The rows of $\hat{B}$ are the columns of $\hat{B}^T$ and after redistribution each process owns those rows of $\hat{B}$ (columns of $\hat{B}^T$) which have the same numbers as the columns of C it owns. Then the algorithm proceeds as in case 2 with $\hat{B}^T$ instead of B, i.e. case 4 of the sequential DGEMM is called, not case 2.

1.4 Results

The implementation of SLAP on an iPSC/860 hypercube was done by assigning one process to each processor. Due to the usage of PARMACS the logical ring of processes was embedded in the hypercube in such a way that neighboring processes in the ring run on neighboring processors in the hypercube. So in cases 1, 2, and 3 where only ring communication is used only nearest neighbor communication in the hypercube is necessary.

For several reasons the parallel DGEMMP needs a lot more memory than the sequential DGEMM. Firstly, we did not want to overwrite the original local part of A (and in case 3 also B) by the received ones, in order to remove the need for additional communication at the end. Secondly, the redistribution routine in case 4 needs some working arrays. Finally, a system communication buffer of the same size as the parts of A and B sent is needed for receiving the data. As explained later (see “Second step of tuning”) asynchronous receive could not be used.

For each number of processors we measured times and MFLOPS for square matrices of sizes from $n = 100$ to the maximum possible size (*maxsize*) for this number of processors by incrementing the size in steps of 100. On one processor *maxsize* was reached at $n = 800$, whereas the limit of DGEMMP on 2 processors was reached at $n = 600$. We therefore do not consider DGEMMP on 2 processors. The maximum sizes were $maxsize = 800$ for 1 and 4 processors, $maxsize = 1200$ for 8 processors, $maxsize = 1800$ for 16 processors and $maxsize = 2700$ for 32 processors.

In addition to the measurements with all three matrices square, we also measured performance with one matrix square and of fixed size and the others rectangular with the size of one dimension varying from 100 to *maxsize* in steps of 100. Although it is possible to make one dimension larger than the above mentioned limits for square matrices when making the other dimension smaller we used the same limits for rectangular matrices as for square ones.

Rectangular matrices were taken into consideration because the parallel routine is not symmetric in A and B. The matrix A is sent around the ring in all cases whereas parts of the matrix B only have to be communicated if B needs to be transposed. Even then the communication pattern for B differs from that for A. The sequential DGEMM was not considered for rectangular matrices.

All measurements were repeated at least three times and the arithmetic mean is shown in the results.

1.4.1 Square matrices

1.4.1.1 Timing the original SLAP routine

The first results for the original SLAP routine showed that there was a considerable communication overhead. The maximum performance achieved was 390 MFLOPS on 32 processors for $n = 2400$ and none of the matrices transposed (see Fig. 3). This is far below the peak performance of 1920 MFLOPS published by the vendor. 60 MFLOPS per node is only a theoretical value: this performance can only be achieved with pipelining, performing one multiplication together with two additions. With a more realistic scenario of one multiplication and one addition

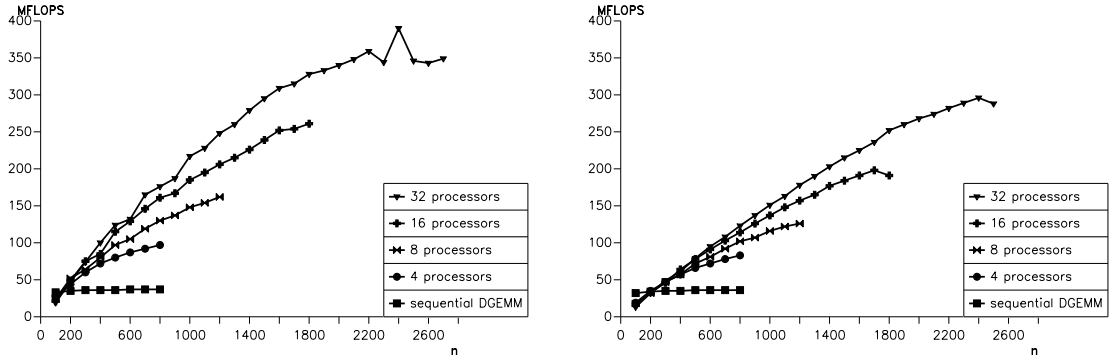

Fig. 3. Performance of the original SLAP version of DGEMMP: Performance on 1, 4, 8, 16, and 32 processors. In the left figure none of the matrices is transposed, in the right figure only *B* is transposed.

the theoretical peak performance is 40 MFLOPS per node. The best performance of DGEMM on one processor is 37 MFLOPS and giving 1184 MFLOPS on 32 processors, which cannot be exceeded. Even compared to this, our result is very inefficient.

For $np = 32$ and $np = 16$ we never reached 50% efficiency. On 8 processors an efficiency of 50% was reached for $n = 1000$ and on 4 processors the matrix size had to be at least 400×400 to reach 50% efficiency (see Fig. 4). In case 3 (*A* not transposed, *B* transposed), where not only *A* but also parts of *B* were sent around the ring, the performance was significantly worse than in all other cases, including the last case, where *B* was redistributed in the beginning (see Fig. 5 left). We think that this is because the redistribution of *B* by the SLAP auxiliary routine needs $\log_2 np$ steps (using the virtual connection of all processors to each other), in each of which $n^2 / 2np$ matrix elements are sent by each processor, whereas in the original DGEMMP algorithm there is additional data to be sent in $np - 1$ steps and the average amount of additional elements is also $n^2 / 2np$. This also explains why the effect becomes more obvious as the number of processors gets larger.

To send parts of *B* along with parts of *A*around the ring, additional system receive-buffer space is needed for case 3, and thus for 32 processors the matrix size limit *maxsize* was $n = 2500$.

1.4.1.2 First step of tuning

Investigation of the code showed that porting SLAP to PARMACS was done in a somewhat “schematic” way. SUPRENUM FORTRAN allows the programmer to send and receive a list of data with array subsections and different types of data just like a standard Fortran i/o list. PARMACS and the iPSC/860 communication require contiguous data in their send command. So the data from SUPRENUM FORTRAN communication lists had to be put into contiguous buffers. Therefore routines were written which copy the data from the list to a buffer of 8000 words, send this buffer when it is full, and copy the next part of the list to the buffer until all elements of the list have been sent. On the receiving side the routines manage the receive request in the same way by receiving buffers of 8000 words, copying the buffer to the places given in

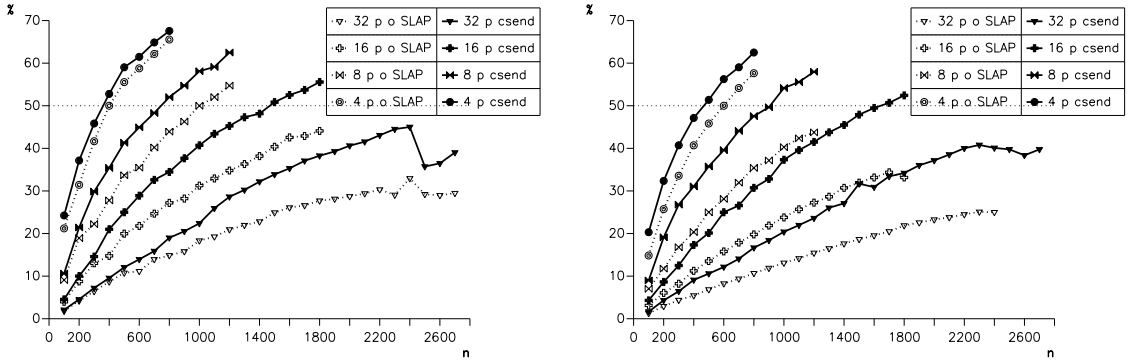

Fig. 4. Efficiency of DGEMMP: Efficiency of DGEMMP original SLAP versus tuned version with *csend*. The left figure shows case 1, none of the matrices transposed, the right figure shows case 3, only *B* transposed.

the receive list and receiving the next buffer. Subsequent calls to a sending or receiving routine can append data to the same buffer if there still is some place left. One input parameter tells when the last call is reached. This made it possible to "translate" implied do-loops in the sending list without starting a new send for each element.

The buffer size of 8000 elements was chosen arbitrarily. If it were made smaller, more communication startups would be necessary for long messages; if it were larger, more space would be statically allocated for this buffer, which could not be used for the rest of the program.

Due to this technique, a lot of sends were started to send large array subsections which could easily be sent as one buffer. As we were familiar with the original SUPRENUM version of the routine we knew that for algorithmic reasons the matrix sections to be sent were copied to a working array before sending them. We had to change this array from two dimensions to one dimension and contigously copy the two-dimensional array sections to the working array. We could then call the PARMACS SEND, which is the same as iPSC *csend*, directly. For 4 processors and $n = 200$ the local part of *A*, which each processor sends, has $200 \cdot 50 = 10000$ elements and so with a buffer of 8000 elements two sends have to be called instead of one. For the largest matrix $800 \cdot 200 = 160000$ elements have to be sent, which leads to 20 sends with a buffer of 8000 elements. On more than 4 processors even more than 20 send and receive latencies could be eliminated in each step, using this approach. In Fig. 4 we see that an efficiency of 50% is now also possible for $np = 16$. For smaller *np* it was achieved with smaller matrices than in the original version. The highest speed was now 530 MFLOPS on 32 processors with *B* not transposed (case 1 and case 2) and $n = 2400$.

The routine for redistributing *B* in case 4 could not be changed in the above mentioned way because the data was not copied to a working array before sending, so there was no workspace to send whole array sections. This routine therefore remained unchanged. Nevertheless, case 3, where parts of *B* were sent instead of redistributing *B* gave lower performance than case 4, so we decided to modify case 3. In the tuned version of case 3, the matrix *B* is redistributed to $\hat{B}$ as in case 4 and then the algorithm proceeds as in case 1, with $\hat{B}^T$ instead of *B*. As a result of

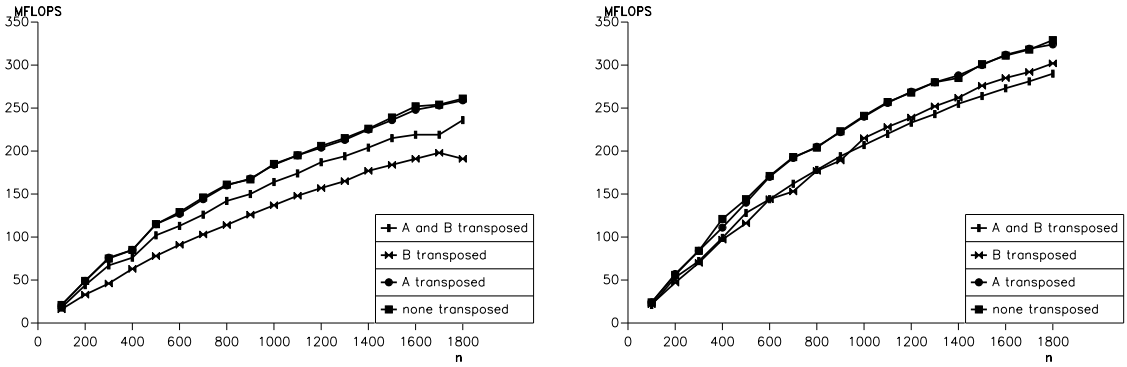

Fig. 5. Performance of DGEMMP on 16 processors: Performance of the original SLAP version (left) and the tuned version using *csend* (right) of DGEMMP on 16 processors of an iPSC/860, comparison of the four cases of transposition of the input matrices.

this, case 3 and case 4 have nearly the same performance, but are of course slower than case 1 and case 2 because of the overhead for the redistribution (see Fig. 5 right).

After this change the system receive buffer in case 3 had the same size as in the other cases and so now it was also possible to measure case 3 up to $n = 2700$ on 32 processors. With the tuned version using *csend*, two new phenomena occurred, which we are not able to explain: For cases 1 and 2, where B is not transposed, we had a significant decrease of performance on 32 processors when the matrix size became larger than 2400. For cases 3 and 4, where the matrix B is redistributed before the computation takes place, this decrease is not as significant as in the first cases, but here the different measurements showed a large variance, especially for case 3. For $n \geq 2400$ the deviations from the mean exceeded 20% whereas for all other measurements the deviations from the mean were between 2% and 10%.

1.4.1.3 Second step of tuning

DGEMMP was designed with the “compute and send ahead” strategy [6] in mind, so we expected a high potential for overlap of communication and computation. This can only be exploited efficiently by the non-blocking version of the send command, *isend*, not by the blocking version, *csend*. Tht current version of PARMACS does not offer a non-blocking send, therefore the original iPSC *isend* has to be used.

Csend blocks the processor until the data to be sent is completely copied to some buffer. With *isend* the processor is only blocked for a short time to set up the copy and send command and continues computing while the communication processor copies the data to the buffer. This can only be used if the data is not changed during the copying process. As DGEMM does not change the matrix A, *isend* can be used to overlap not only the time for moving the data to the other processor but also the time for copying the data to the buffer with computation.

The asynchronous receive operation *irecv* for the iPSC puts the received data immediately into the desired location instead of a receive buffer and thereby saves one copy operation. It over-

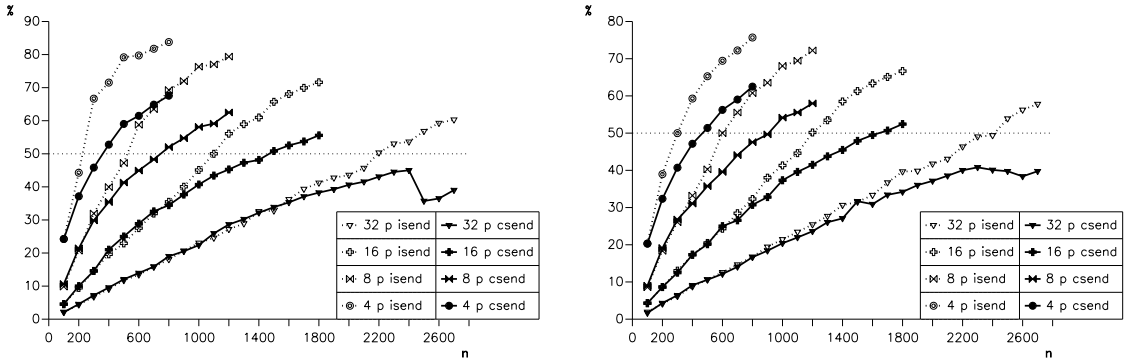

Fig. 6. Efficiency of DGEMMP: Efficiency of DGEMMP tuned version with *csend* versus tuned version with *isend*. The left figure shows case 1, none of the matrices transposed, the right figure shows case 3, only *B* transposed.

writes this location as the message arrives and can not be used if this location is read during this time. The matrix *A* is read during the call to DGEMM, so it is not possible to post an *irecv* for the next part of *A* before the call to DGEMM. So the *crecv* after the call to DGEMM remained unchanged. To take advantage of *irecv* in addition to *isend*, a complete restructuring of the code would be necessary, which is beyond the scope of this paper.

Nevertheless, for large matrices even the usage of *isend* alone still leads to better performance especially for small *np*. For 4 processors an efficiency of more than 80% was achieved (see Fig. 6). The maximum performance on 32 processors now reached 724 MFLOPS for $n = 2700$ and case 2. The performance for $n > 2400$ did no longer decrease and the individual timing measurements for the same case and matrix size deviated less from the arithmetic mean than for the tuned version using *csend*, despite the fact that the redistribution routine remained unchanged.

1.4.2 Rectangular matrices

In our timings we always kept two of the dimensions fixed and varied the other from 100 to *maxsize* (see “Results”). So *C* was always rectangular with variable size whereas either *B* or *A* remained square with fixed size *k*. For $np = 4$ we had $k = 400$ and $k = 800$, for $np = 8$ we had $k = 600, 1200$, for $np = 16$ we had $k = 900, 1800$, and for $np = 32$ we had $k = 900, 1800, 2700$.

1.4.2.1 Matrix A rectangular of variable size, B square of fixed size

In this case *m* is varied from 100 to *maxsize* whereas $k = n$ = fixed. The number of arithmetic operations on each processor is $O(m \cdot k^2)/np$.

If *B* requires no transposition, the total number of elements to be communicated by each processor is $O(m \cdot k)$. So the computation to communication ratio on each processor is $O(k/np)$.

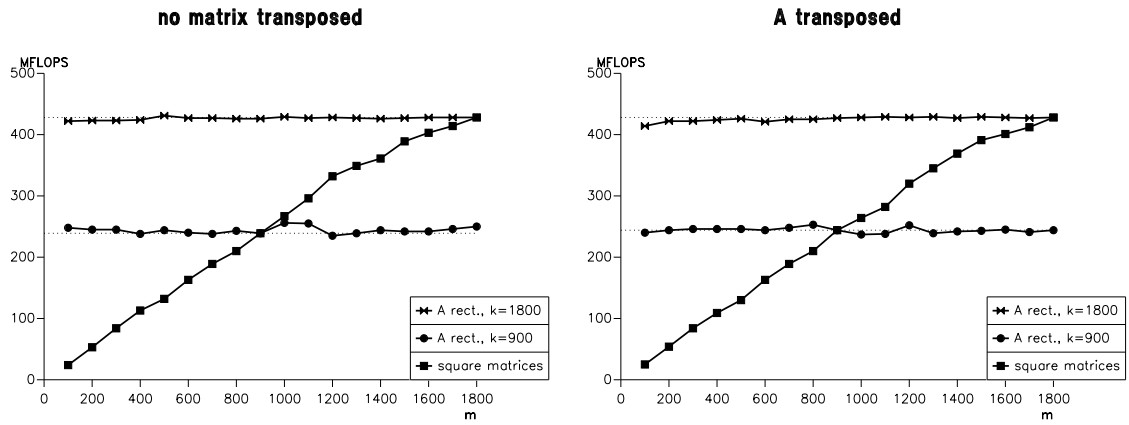

Fig. 7. Performance on 16 processors, A rectangular, B square and not transposed: Comparison between all matrices square and matrix *B* square of fixed size and *A* rectangular of variable size.

This is the same ratio as for $A, B, C \in \mathbb{R}^{k,k}$. Indeed for all *m*, the performance in MFLOPS is in this case nearly the same as for $A, B, C \in \mathbb{R}^{k,k}$ (see Fig. 7).

If *B* has to be transposed, there is an additional overhead of communicating $O(k^2 \log_2 np/2np)$ elements for the redistribution of *B* on each processor. This is the same amount of communication as for $A, B, C \in \mathbb{R}^{k,k}$. The computation to communication ratio now is

$$O\left(\frac{k}{np(1+\log_2 np/2np)}\right) \quad \text{for} \quad A,\ B,\ C \in \mathbb{R}^{k,k} \quad \text{and}$$

$$O\left(\frac{k}{np(1+k/m\cdot \log_2 np/2np)}\right) \quad \text{for} \quad A^{[T]},\ C \in \mathbb{R}^{m,k},\quad B \in \mathbb{R}^{k,k}.$$

This means that for $A^{[T]}, C \in \mathbb{R}^{m,k}$ the computation to communication ratio is smaller than for $A, B, C \in \mathbb{R}^{k,k}$ for $m < k$ and higher for $m > k$. For $np \le 16$ and $k = maxsize/2$ the higher computation to communication ratio is visible, as the MFLOPS rates still increase slightly for $m > k$ (see Fig. 8). For $np = 32$ and $k = 900$ or $k = 1800$ this increase can no longer be seen (see Fig. 9). We think this is due to the factor $\log_2 np/2np$ in the amount of additional communication which rapidly decreases with an increasing number of processors. The denominator varies only slightly with *m*, especially for $m > k$. So for 32 processors the additional communication has only slight influence on the performance. A small reduction of this additional communication no longer causes a significant increase of performance.

We are not able to explain why, for all numbers of processors, cases 3 and 4 behave so differently for small *m*. We measured the single-node performance for those matrices which have to be multiplied on each processor for $np = 8$, $n = k = 1200$, $m = 100, 200, 300, 400$ and arrived at a figure of 34 to 35 MFLOPS in cases 1, 2, and 3, but only 16 MFLOPS for $m = 100$, 23 MFLOPS for $m = 200$, 26 MFLOPS for $m = 300$, and 28 MFLOPS for $m = 400$ in case 4. On the other hand cases 3 and 4 behave similarly for square matrices, although the strange behaviour of the sequential routine in case 4 could also be seen, to a lesser

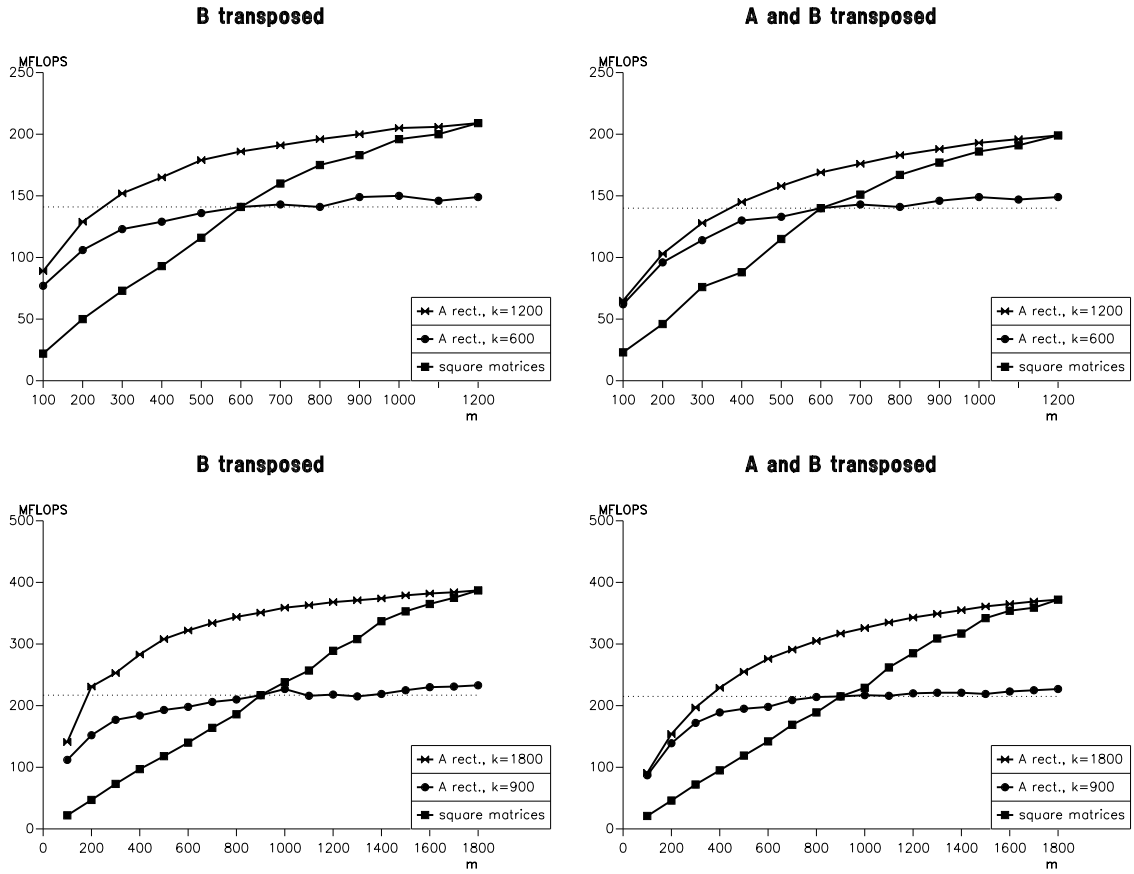

Fig. 8. Performance on 8 (upper) and 16 (lower) processors, A rectangular, B square and transposed: Comparison between all matrices square and matrix *B* square of fixed size and *A* rectangular of variable size.

extent, for those matrices which have to be multiplied on a single processor for $np = 8$ and square matrices of size $m = n = k = 100, \ldots, 400$ (see Tab. 1 and Fig. 8).

k	**case 1** [**MFLOPS**]	**case 2** [**MFLOPS**]	**case 3** [**MFLOPS**]	**case 4** [**MFLOPS**]
100	19	22	19	12
200	24	29	24	20
300	29	31	29	23
400	29	31	29	26

Tab. 1. : The single-node MFLOPS rates for those matrices which have to be multiplied on a single node for global square matrices of size *k*.

1.4.2.2 Matrix A square of fixed size, B rectangular of variable size

In this case *n* is varied from 100 to *maxsize* whereas $m = k$ = fixed. The number of arithmetic operations on each processor is $O(n \cdot k^2)/np$.

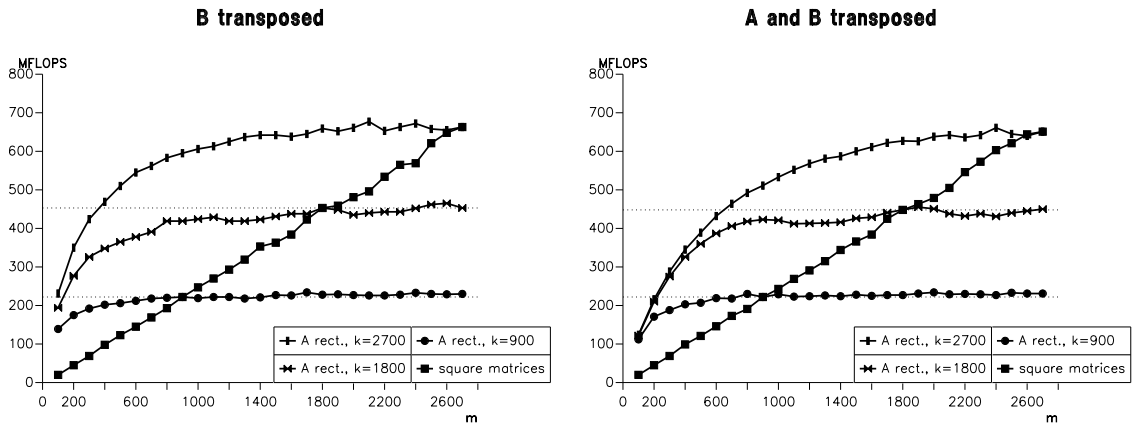

Fig. 9. Performance on 32 processors, A rectangular, B square and transposed: Comparison between all matrices square and matrix *B* square of fixed size and *A* rectangular of variable size.

If *B* requires no transposition the total amount of elements to be communicated by each processor is $O(k^2)$. So the computation to communication ratio on each processor is $O(n/np)$. This is the same ratio as for $A, B, C \in \mathbb{R}^{n,n}$. Indeed, in those cases the performance is almost the same for $A, B, C \in \mathbb{R}^{n,n}$ as for $A \in \mathbb{R}^{k,k}$ and $B, C \in \mathbb{R}^{k,n}$ (see Fig. 10).

If *B* has to be transposed, additional communication for the redistribution of *B* is required again. Now the surplus amount is $O(k \cdot n \cdot \log_2 np/2np)$. The computation to communication ratio is

$$O\left(\frac{n}{np(1 + n/k \cdot \log_2 np/2np)}\right) \quad \text{for} \quad B^T,\ C \in \mathbb{R}^{k,n},\ A \in \mathbb{R}^{k,k}.$$

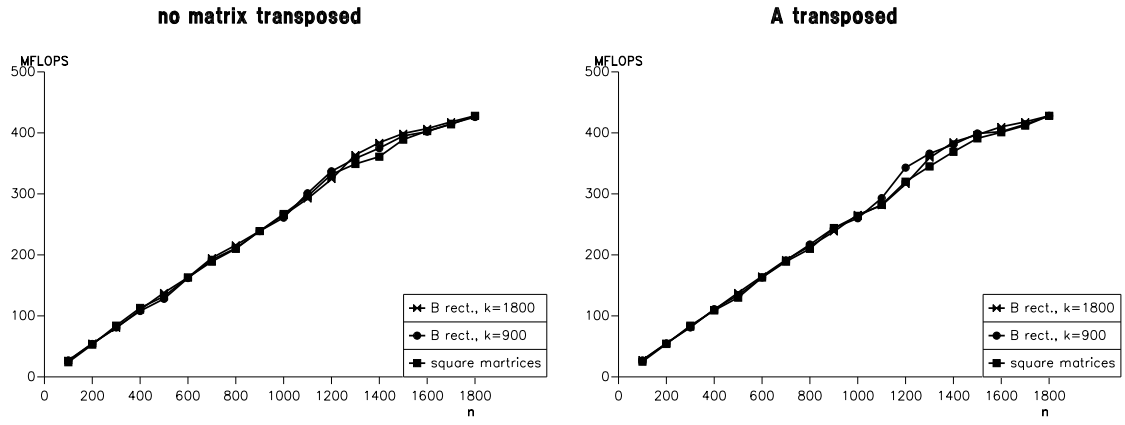

Fig. 10. Performance on 16 processors, A square, B rectangular and not transposed: Comparison between all matrices square and matrix *A* square of fixed size and *B* rectangular of variable size.

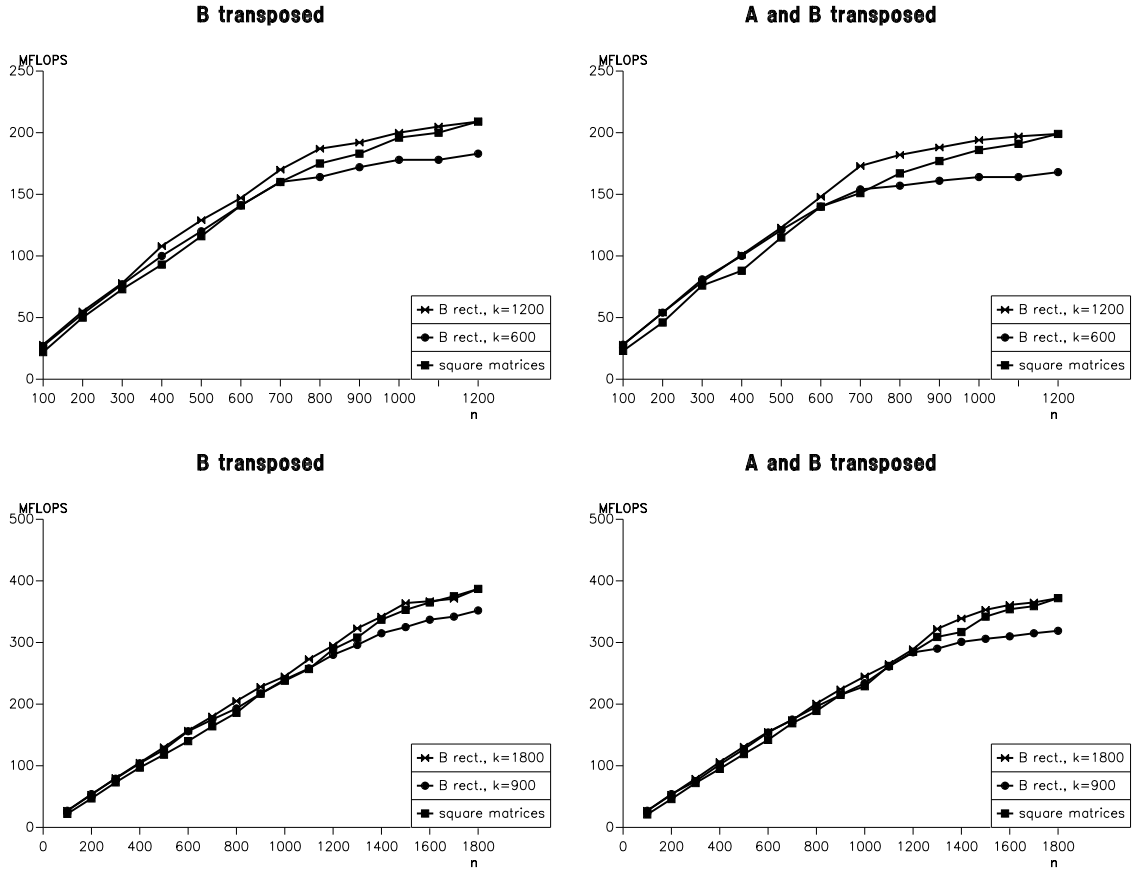

Fig. 11. Performance on 8 (upper) and 16 (lower) processors, A square, B rectangular and transposed: Comparison between all matrices square and matrix *A* square of fixed size and *B* rectangular of variable size.

It is higher than for $A,\ B,\ C \in \mathbb{R}^{n,n}$ if $n < k$ and smaller if $n > k$. The effect can again be seen for $np \leq 16$ (see Fig. 11) whereas for $np = 32$ only the decreasing performance for large n and $k = 900$ is visible (see Fig. 12).

1.5 Conclusions

For a parallel library routine, which is tuned slightly for the target architecture, acceptable performance up to a maximum of 724 MFLOPS can be achieved on 32 processors. For the matrix multiplication routine DGEMMP the computation to communication ratio still has great influence on the performance. We suppose this is at least partially due to the use of *crecv*, which includes copying the received data from the receive buffer to the desired location. Restructuring the routine to use *irecv*, allowing the receive operation to also be overlapped with computation, could lead to still better performance for large matrices. For small matrices the amount of computation may not be large enough to overlap all communication.

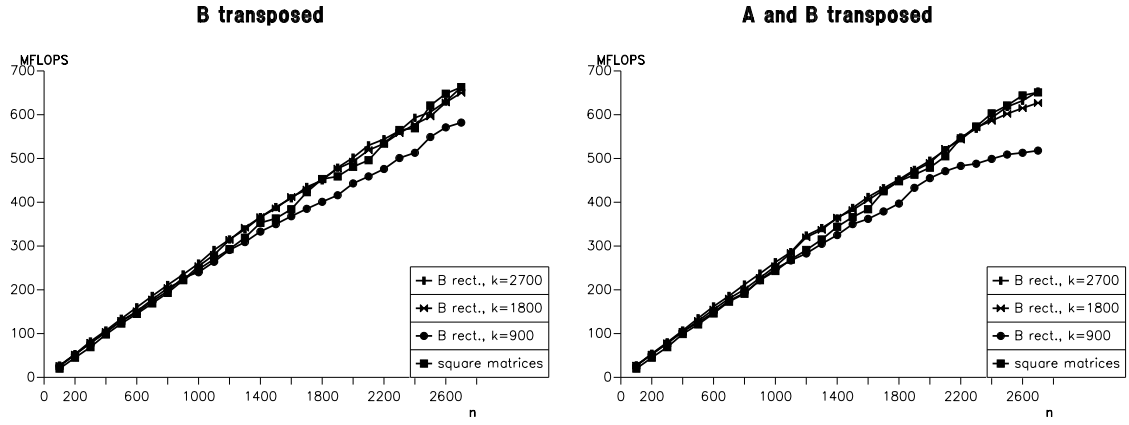

Fig. 12. Performance on 32 processors, A square, B rectangular and transposed: Comparison between all matrices square and matrix *A* square of fixed size and *B* rectangular of variable size.

From Fig. 6 we see that the version of DGEMMP using *isend* becomes more efficient than the version using *csend* if the matrix size n is large enough for the number of processors. On 4 processors n has to be at least 200, on 8 processeors n has to be greater or equal 400, on 16 processors $n \geq 700$ is necessary, and on 32 processors $n \geq 1600$ is the limit for *isend* to be more efficient than *csend*. This suggests that n/np has to be larger than 50 in order to see the effect of overlapping the send and computation. We assume the ratio has to be larger still, in order to overlap also the receive operations by computation.

1.6 Acknowledgement

We would like to thank W. Rönsch, now IBM Scientific Center Heidelberg, the author of the original SLAP routine DGEMMP and the redistribution routine mentioned in this paper for his very useful comments.

1.7 References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, LAPACK Users' Guide, SIAM Philadelphia, 1992

[2] L. Bomans, D. Roose, R. Hempel, The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2, Parallel Computing 15 (1990), 119-132

[3] R. Calkin, R. Hempel, H.-C. Hoppe, and P. Wypior, Portrable Programming with the PARMACS Message-Passing Library, to appear in Parallel Computing 19 (1993)

[4] J.J. Dongarra, J.J. Du Croz, S.J. Hammarling, and R.J. Hanson, An extended Set of Fortran Basic Linear Algebra Subprograms, with – Model Implementation and Test Programs, ACM Trans. Math. Software 14 (1988), 1-32

[5] J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, A Set of Level 3 Basic Linear Algebra Subprograms, Argonne National Laboratory, Mathematics and Computer Science Division, Preprint No. 1, August 1988

[6] I. Gutheil, W. Rönsch, and H. Strauß, Lineare Algebra Software für SUPRENUM, Forschungszentrum Jülich: Bericht Nr. 2345, Jülich 1990

[7] iPSC/860 FORTRAN SYSTEM CALLS REFERENCE MANUAL, (312234-001), Intel Corporation, 1992

[8] Kuck & Associates, CLASSPACK Basic Math Library User's Guide, Release 1.2, Document 9202003, Champaign, IL, 1992

[9] PALLAS Gesellschaft für Parallele Anwendungen und Systeme mbH, PARMACS, Manual (1992)

[10] PALLAS Gesellschaft für Parallele Anwendungen und Systeme mbH, SLAP Scientific Linear Algebra Package, Manual (1991)

[11] W. Rönsch and H. Strauß, A Linear Algebra Package for a Local Memory Multiprocessor: Problems, Proposals and Solution, Parallel Computing 7 (1988), 413-418

[12] W. Rönsch and H. Strauß, Design Aspects of a Linear Algebra Package for the SUPRENUM Multiprocessor System, in J. Dongarra, I. Duff, P. Gaffney, and S. McKee eds., Vector and Parallel Computing, issues in applied research and development, (Ellis Horwood Ltd. Publ.) (1989) 299-315

[13] U. Trottenberg, Guest Editor, Proceedings of the 2nd International SUPRENUM Colloquium 30 September - 2 October 1987, Bonn, Fed. Rep. Germany, Parallel Computing 7 (1988), 263-499